Incremental Low-High Orders of Directed Graphs and Applications*

Loukas Georgiadis[1], Konstantinos Giannis[2], Aikaterini Karanasiou[3], and Luigi Laura[4]

1 **Department of Computer Science & Engineering, University of Ioannina, Ioannina, Greece**
loukas@cs.uoi.gr
2 **Department of Computer Science & Engineering, University of Ioannina, Ioannina, Greece**
giannis_konstantinos@outlook.com
3 **Università di Roma "Tor Vergata", Rome, Italy**
aikaranasiou@gmail.com
4 **"Sapienza" Università di Roma, Rome, Italy**
laura@dis.uniroma1.it

Abstract

A flow graph $G = (V, E, s)$ is a directed graph with a distinguished start vertex s. The dominator tree D of G is a tree rooted at s, such that a vertex v is an ancestor of a vertex w if and only if all paths from s to w include v. The dominator tree is a central tool in program optimization and code generation, and has many applications in other diverse areas including constraint programming, circuit testing, biology, and in algorithms for graph connectivity problems. A low-high order of G is a preorder δ of D that certifies the correctness of D, and has further applications in connectivity and path-determination problems.

In this paper we consider how to maintain efficiently a low-high order of a flow graph incrementally under edge insertions. We present algorithms that run in $O(mn)$ total time for a sequence of edge insertions in a flow graph with n vertices, where m is the total number of edges after all insertions. These immediately provide the first incremental *certifying algorithms* for maintaining the dominator tree in $O(mn)$ total time, and also imply incremental algorithms for other problems. Hence, we provide a substantial improvement over the $O(m^2)$ straightforward algorithms, which recompute the solution from scratch after each edge insertion. Furthermore, we provide efficient implementations of our algorithms and conduct an extensive experimental study on real-world graphs taken from a variety of application areas. The experimental results show that our algorithms perform very well in practice.

1998 ACM Subject Classification E.1 [Data Structures] Graphs and Networks, Lists, Stacks, and Queues, Trees, G.2.2 [Graph Theory] Graph Algorithms

Keywords and phrases connectivity, directed graphs, dominators, dynamic algorithms

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.27

1 Introduction

A *flow graph* $G = (V, E, s)$ is a directed graph (digraph) with a distinguished start vertex $s \in V$. A vertex v is *reachable* in G if there is a path from s to v; v is *unreachable* if no

* A full version of the paper is available at http://arxiv.org/abs/1608.06462.

16th International Symposium on Experimental Algorithms (SEA 2017).
Editors: Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman; Article No. 27; pp. 27:1–27:21

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

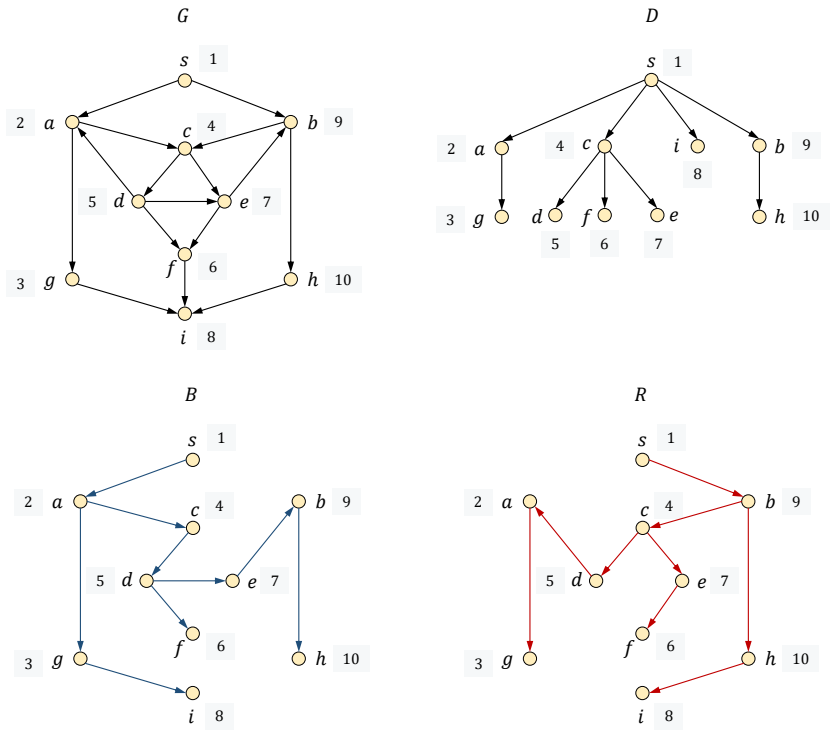

Figure 1 A flow graph G, its dominator tree D, and two strongly divergent spanning trees B and R. The numbers correspond to a preorder numbering of D that is a low-high order of G.

such path exists. The *dominator relation* in G is defined for the set of reachable vertices as follows. A vertex v is a *dominator* of a vertex w (v *dominates* w) if every path from s to w contains v; v is a *proper dominator* of w if v dominates w and $v \neq w$. The dominator relation in G can be represented by a tree rooted at s, the *dominator tree* D, such that v dominates w if and only if v is an ancestor of w in D. If $w \neq s$ is reachable, we denote by $d(w)$ the parent of w in D. Lengauer and Tarjan [34] presented an algorithm for computing dominators in $O(m\alpha(m,n))$ time for a flow graph with n vertices and m edges, where α is a functional inverse of Ackermann's function [44]. Subsequently, several linear-time algorithms were discovered [3, 10, 15, 16]. The dominator tree is a central tool in program optimization and code generation [12], and it has applications in other diverse areas including constraint programming [40], circuit testing [5], theoretical biology [2], memory profiling [36], the analysis of diffusion networks [28], and in connectivity problems [18, 19, 21, 20, 23, 29, 30, 31, 32].

A *low-high order* δ *of* G [25] is a preorder of the dominator tree D such for all reachable vertices $v \neq s$, $(d(v), v) \in E$ or there are two edges $(u, v) \in E$, $(w, v) \in E$ such that u and w are reachable, u is less than v ($u <_\delta v$), v is less than w ($v <_\delta w$), and w is not a descendant of v in D. See Figure 1. Every flow graph G has a low-high order, computable in linear-time [25]. Low-high orders provide a correctness certificate for dominator trees that is straightforward to verify [47]. By augmenting an algorithm that computes the dominator tree D of a flow graph G so that it also computes a low-high order of G, one obtains a *certifying algorithm* to compute D. (A *certifying algorithm* [37] outputs both the solution and a correctness certificate, with the property that it is straightforward to use the certificate to verify that the computed solution is correct.) Low-high orders also have applications in path-determination problems [46] and in fault-tolerant network design [6, 7, 26].

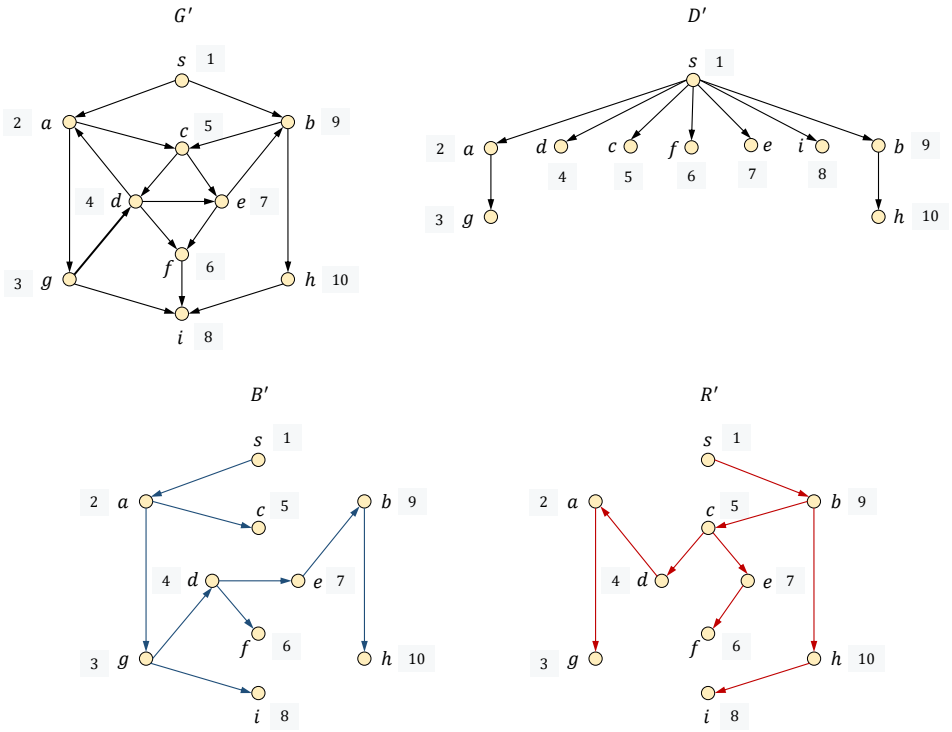

Figure 2 The flow graph of Figure 1 after the insertion of edge (g, d), and its updated dominator tree D' with a low-high order, and two strongly divergent spanning trees B' and R'.

A notion closely related to low-high orders is that of divergent spanning trees [25]. Let V_r be the set of reachable vertices, and let $G[V_r]$ be the flow graph with start vertex s that is induced by V_r. Two spanning trees B and R of $G[V_r]$, rooted at s, are *divergent* if for all v, the paths from s to v in B and R share only the dominators of v; B and R are *strongly divergent* if for every pair of vertices v and w, either the path in B from s to v and the path in R from s to w share only the common dominators of v and w, or the path in R from s to v and the path in B from s to w share only the common dominators of v and w. In order to simplify our notation, we will refer to B and R, with some abuse of terminology, as strongly divergent spanning trees of G. Every flow graph has a pair of strongly divergent spanning trees. Given a low-high order of G, it is straightforward to compute two strongly divergent spanning trees of G in $O(m)$ time [25]. Divergent spanning trees can be used in data structures that compute pairs of vertex-disjoint s-t paths in 2-vertex connected digraphs (for any two query vertices s and t) [18], in fast algorithms for approximating the smallest 2-vertex-connected spanning subgraph of a digraph [19], and in constructing sparse subgraphs of a given digraph that maintain certain connectivity requirements [21, 31, 32].

In this paper we consider how to update a low-high order of a flow graph through a sequence of edge insertions. See Figure 2. The difficulty in updating the dominator tree and a low-high order is due to the following facts. An affected vertex can be arbitrarily far from the inserted edge, and a single edge insertion may cause $O(n)$ parent changes in D. Furthermore, since a low-high order is a preorder of D, a single edge insertion may cause $O(n)$ changes in this order, even if there is only one vertex that is assigned a new parent in D after the insertion. More generally, we note that the hardness of dynamic algorithms on digraphs

has been recently supported also by conditional lower bounds [1]. Our first contribution is to show that we can maintain a low-high order of a flow graph G with n vertices through a sequence of edge insertions in $O(mn)$ total time, where m is the total number of edges after all insertions. Hence, we obtain a substantial improvement over the naive solution of recomputing a low-high order from scratch after each edge insertion, which takes $O(m^2)$ total time. Our result also implies the first incremental *certifying algorithms* [37] for computing dominators in $O(mn)$ total time, which answers an open question in [25]. We present two algorithms that achieve this bound, a simple algorithm based on sparsification and a more sophisticated algorithm. Both algorithms combine the incremental dominators algorithm of [22] with the linear-time computation of two divergent spanning trees from [25]. Our sophisticated algorithm also applies a slightly modified version of a static low-high algorithm from [25] on an auxiliary graph. We remark that the incremental dominators problem arises in various applications, such as incremental data flow analysis and compilation [11, 17, 41, 42], distributed authorization [38], and in incremental algorithms for maintaining 2-connectivity relations in directed graphs [23]. We present some applications of our result on incremental low-high order maintenance to incremental connectivity problems in Appendix A.

We assess the merits of our algorithm in practical scenarios by conducting a thorough experimental study with graphs taken from a variety of application areas. Although both the sparsification algorithm and the sophisticated algorithm have the same worst-case running time, our experimental results show that a carefully engineered implementation of the latter is by far superior in practice.

For lack of space, some proofs are omitted from this extended abstract. They are provided in the full version [24].

2 Preliminaries

Let $G = (V, E, s)$ be a flow graph with start vertex s, and let D be the dominator tree of G. A spanning tree T of G is a tree with root s that contains a path from s to v for all reachable vertices v. We refer to a spanning subgraph F of T as a spanning forest of G. Given a rooted tree T, we denote by $T(v)$ the subtree of T rooted at v (we also view $T(v)$ as the set of descendants of v). Let T be a tree rooted at s with vertex set $V_T \subseteq V$, and let $t(v)$ denote the parent of a vertex $v \in V_T$ in T. If v is an ancestor of w, $T[v, w]$ is the path in T from v to w. In particular, $D[s, v]$ consists of the vertices that dominate v. If v is a proper ancestor of w, $T(v, w]$ is the path to w from the child of v that is an ancestor of w. Tree T is *flat* if its root is the parent of every other vertex. Suppose now that the vertex set V_T of T consists of the vertices reachable from s. Tree T has the *parent property* if for all $(v, w) \in E$ with v and w reachable, v is a descendant of $t(w)$ in T. If T has the parent property and has a low-high order, then $T = D$ [25]. For any vertex $v \in V$, we denote by $C(v)$ the set of children of v in D. A *preorder* of T is a total order of the vertices of T such that, for every vertex v, the descendants of v are ordered consecutively, with v first. Let ζ be a preorder of D. Consider a vertex $v \neq s$. We say that ζ *is a low-high order for* v *in* G, if $(d(v), v) \in E$ or there are two edges $(u, v) \in E$, $(w, v) \in E$ such that $u <_\zeta v$ and $v <_\zeta w$, and w is not a descendant of v in D. Given a graph $G = (V, E)$ and a set of edges $S \subseteq V \times V$, we denote by $G \cup S$ the graph obtained by inserting into G the edges of S.

3 Incremental low-high order

In this section we describe two algorithms to maintain a low-high order of a digraph through a sequence of edge insertions. We first review some useful facts for updating a dominator

tree after an edge insertion [4, 22, 41]. Let (x, y) be the edge to be inserted. We consider the effect of this insertion when both x and y are reachable. Let G' be the flow graph that results from G after inserting (x, y). Similarly, if D is the dominator tree of G before the insertion, we let D' be the dominator tree of G'. Also, for any function f on V, we let f' be the function after the update. We say that vertex v is *affected* by the update if $d(v)$ (its parent in D) changes, i.e., $d'(v) \neq d(v)$. We let A denote the set of affected vertices. Note that we can have $D'[s, v] \neq D[s, v]$ even if v is not affected. We let $nca(x, y)$ denote the nearest common ancestor of x and y in the dominator tree D. We also denote by $depth(v)$ the depth of a reachable vertex v in D. There are affected vertices after the insertion of (x, y) if and only if $nca(x, y)$ is not a descendant of $d(y)$ [41]. A characterization of the affected vertices is provided by the following lemma, which is a refinement of a result in [4].

▶ **Lemma 1** ([22]). *Suppose x and y are reachable vertices in G. A vertex v is affected after the insertion of edge (x, y) if and only if $depth(nca(x, y)) < depth(d(v))$ and there is a path π in G from y to v such that $depth(d(v)) < depth(w)$ for all $w \in \pi$. If v is affected, then it becomes a child of $nca(x, y)$ in D', i.e., $d'(v) = nca(x, y)$.*

The algorithm (DBS) in [22] applies Lemma 1 to identify affected vertices by starting a search from y (if y is not affected, then no other vertex is). To do this search for affected vertices, it suffices to maintain the outgoing and incoming edges of each vertex. These sets are organized as singly linked lists, so that a new edge can be inserted in $O(1)$ time. The dominator tree D is represented by the parent function d. We also maintain the depth in D of each reachable vertex. We say that a vertex v is *scanned*, if the edges leaving v are examined during the search for affected vertices, and that it is *visited* if there is a scanned vertex u such that (u, v) is an edge in G. By Lemma 1, a visited vertex v is scanned if $depth(nca(x, y)) < depth(d(v))$.

▶ **Lemma 2** ([22]). *Let v be a scanned vertex. Then v is a descendant of an affected vertex in D.*

3.1 Sparsification Algorithm

In this algorithm we maintain, after each insertion, a subgraph $H = (V, E_H)$ of G with $O(n)$ edges that has the same dominator tree as G. Then, we can compute a low-high order δ of H in $O(|E_H|) = O(n)$ time. Note that by the definition of H, δ is also a valid low-high order of G. An edge insertion is processed by the routine SparseInsertEdge, shown below. Subgraph H is formed by the edges of two divergent spanning trees B and R of G. After the insertion of an edge (x, y), where both x and y are reachable, we form a graph H' by inserting into H a set of edges $Last(A)$ found during the search for the set of affected vertices A. Specifically, $Last(A)$ contains edge (x, y) and, for each affected vertex $v \neq y$, the last edge on a path π_{yv} that satisfies Lemma 1. Then, we set $H' = H \cup Last(A)$. Finally, we compute a low-high order and two divergent spanning trees of H', which are also valid for G'. We can show that this algorithm runs in $O(mn)$ total time.

3.2 Local Low-High Order Algorithm

Here we develop a more sophisticated and more practical algorithm that maintains a low-high order δ of a flow graph $G = (V, E, s)$ through a sequence of edge insertions. Our algorithm uses the incremental dominators algorithm of [22] to update the dominator tree D of G after each edge insertion. We describe a process to update δ based on the relation among vertices in D that are affected by the insertion. This enables us to identify a subset of vertices for

Algorithm 1: SparseInsertEdge(G, D, δ, B, R, e).

Input: Flow graph $G = (V, E, s)$, its dominator tree D, a low-high order δ of G, two divergent spanning trees B and R of G, and a new edge $e = (x, y)$.

Output: Flow graph $G' = (V, E \cup (x, y), s)$, its dominator tree D', a low-high order δ' of G', and two divergent spanning trees B' and R' of G'.

```
1  Insert e into G to obtain G'.
2  if x is unreachable in G then return (G', D, δ, B, R)
3  else if y is unreachable in G then
4  |   Compute the dominator tree D', two divergent spanning trees B' and R', and a
   |    low-high order δ' of G'.
5  |   return (G', D', δ', B', R')
6  end
7  Let H = B ∪ R.
8  Compute the updated dominator tree D' of G' and return a list A of the affected
     vertices, and a list Last(A) of the last edge entering each v ∈ A in a path satisfying
     Lemma 1.
9  Compute the subgraph H' = H ∪ Last(A) of G'.
10 Compute the dominator tree D', two divergent spanning trees B' and R', and a
     low-high order δ' of H'.
11 return (G', D', δ', B', R')
```

which we can compute a "local" low-high order, that can be extended to a valid low-high order of G after the update. We show that such a "local" low-high order can be computed by a slightly modified version of an algorithm from [25]. We apply this algorithm on a sufficiently small flow graph that is defined by the affected vertices, and is constructed using the concept of derived edges [45].

3.2.1 Derived edges and derived flow graphs

Derived graphs, first defined in [45], reduce the problem of finding a low-high order to the case of a flat dominator tree [25]. By the parent property of D, if (v, w) is an edge of G, the parent $d(w)$ of w is an ancestor of v in D. Let (v, w) be an edge of G, with w not an ancestor of v in D. Then, the *derived edge* of (v, w) is the edge $(\overline{v}, w)$, where $\overline{v} = v$ if $v = d(w)$, $\overline{v}$ is the sibling of w that is an ancestor of v if $v \neq d(w)$. If w is an ancestor of v in D, then the derived edge of (v, w) is null. Note that a derived edge $(\overline{v}, w)$ may not be an original edge of G. For any vertex $w \in V$ such that $C(w) \neq \emptyset$, we define the *derived flow graph of* w, denoted by $G_w = (V_w, E_w, w)$, as the flow graph with start vertex w, vertex set $V_w = C(w) \cup \{w\}$, and edge set $E_w = \{(\overline{u}, v) \mid v \in V_w \text{ and } (\overline{u}, v) \text{ is the non-null derived edge of some edge in } E\}$. By definition, G_w has flat dominator tree, that is, w is the only proper dominator of any vertex $v \in V_w \setminus \{w\}$. We can compute a low-high order δ of G by computing a low-high order δ_w in each derived flow graph G_w. Given these low-high orders δ_w, we can compute a low-high order of G in $O(n)$ time by a depth-first traversal of D. During this traversal, we visit the children of each vertex w in their order in δ_w, and number the vertices from 1 to n as they are visited. The resulting preorder of D is low-high on G. Our incremental algorithm identifies, after each edge insertion, a specific derived flow graph G_w for which a low-high order δ_w needs to be updated. Then, it uses δ_w to update the low-high order of the whole flow graph G. Still, computing a low-high order of G_w can be too expensive to give us the

desired running time. Fortunately, we can overcome this obstacle by exploiting a relation among the vertices that are affected by the insertion, as specified below. This allows us to compute δ_w in a contracted version of G_w.

3.2.2 Affected vertices

Let (x, y) be the inserted vertex, where both x and y are reachable. Consider the execution of algorithm DBS [22] that updates the dominator tree by applying Lemma 1. Suppose vertex v is scanned, and let q be the nearest affected ancestor of v in D. Then, by Lemma 1, vertex q is a child of $nca(x, y)$ in D', i.e., $d'(q) = nca(x, y)$, and v remains a descendant of q in D'.

Our next lemma provides a key result about the relation of the affected vertices in D.

▶ **Lemma 3.** *All vertices that are affected by the insertion of (x, y) are descendants of a common child c of $nca(x, y)$.*

We shall apply Lemma 3 to construct a flow graph G_A for the affected vertices. Then, we shall use G_A to compute a "local" low-high order that we extend to a valid low-high order of G'.

3.2.3 Low-high order augmentation

Let δ be a low-high order of G, and let δ' be a preorder of the dominator tree D' of G'. We say that δ' *agrees with* δ if the following condition holds for any pair of siblings u, v in D that are not affected by the insertion of (x, y): $u <_{\delta'} v$ if and only if $u <_{\delta} v$. We can show that there is a low-high order δ' of G' that agrees with δ. Moreover, we have the following result:

▶ **Lemma 4.** *Let δ' be a preorder of D' that agrees with δ. Let v be a vertex that is not a child of $nca(x, y)$ and is not affected by the insertion of (x, y). Then δ' is a low-high order for v in G'.*

We can apply Lemmata 1 and 4 to show that in order to compute a low-high order of G', it suffices to compute a low-high order for the derived flow graph G'_z, where $z = nca(x, y)$. Still, the computation of a low-high order of G'_z is too expensive to give us the desired running time. Fortunately, as we show next, we can limit these computations for a contracted version of G'_z, defined by the affected vertices.

Let δ be a low-high order of G before the insertion of (x, y). Also, let $z = nca(x, y)$, and let δ_z be a corresponding low-high order of the derived flow graph G_z. That is, δ_z is the restriction of δ to z and its children in D. Consider the child c of z that, by Lemma 3, is an ancestor of all the affected vertices. Let α and β, respectively, be the predecessor and successor of c in δ_z. Note that α or β may be null. An *augmentation of* δ_z is an order δ'_z of $C'(z) \cup \{z\}$ that results from δ_z by inserting the affected vertices arbitrarily around c, that is, each affected vertex is placed in an arbitrary position between α and c or between c and β.

▶ **Lemma 5.** *Let $z = nca(x, y)$, and let δ_z be a low-high order of the derived flow graph G_z before the insertion of (x, y). Also, let δ'_z be an augmentation of δ_z, and let δ' be a preorder of D' that extends δ'_z. Then, for each child v of z in D, δ' is a low-high order for v in G'.*

3.2.4 Algorithm

Now we are ready to describe our incremental algorithm for maintaining a low-high order δ of G. For each vertex v that is not a leaf in D, we maintain a list of its children $C(v)$ in D, ordered by δ. Also, for each vertex $v \neq s$, we keep two variables $low(v)$ and $high(v)$.

Algorithm 2: LocalInsertEdge$(G, D, \delta, mark, low, high, e)$.

Input: Flow graph $G = (V, E, s)$, its dominator tree D, a low-high order δ of G, arrays *mark*, *low* and *high*, and a new edge $e = (x, y)$.

Output: Flow graph $G' = (V, E \cup (x, y), s)$, its dominator tree D', a low-high order δ' of G', and arrays *mark*$'$, *low*$'$ and *high*$'$.

```
1  Insert e into G to obtain G'.
2  if x is unreachable in G then return (G', D, δ, mark, low, high)
3  else if y is unreachable in G then
4  |   Compute the dominator tree D' and a low-high order δ' of G', together with the
   |    corresponding arrays mark', low', and high'.
5  |   return (G', D', δ', mark', low', high')
6  end
7  Compute the nearest common ancestor z of x and y in D.
8  Compute the updated dominator tree D' of G' and return a list A of the affected
    vertices.
9  foreach vertex v ∈ A do mark'(y) ← false
10 if z = x then mark'(y) ← true
11 Compute a low-high order ζ of the derived affected flow graph G_A.
12 Compute the updated low-high order δ' of G' by ordering the vertices in A ∪ {c}
    according to ζ.
13 foreach vertex v ∈ A ∪ {c} do
14 |   find edges (u, v) and (w, v) such that u <_δ' v <_δ' w and w ∉ D'(v)
15 |   set low'(v) ← u and high'(v) ← w
16 end
17 return (G', D', δ', mark', low', high')
```

Variable $low(v)$ stores an edge (u, v) such that $u \neq d(v)$ and $u <_\delta v$; $low(v) = null$ if no such edge exists. Similarly, $high(v)$ stores an edge (w, v) such that and $v <_\delta w$ and w is not a descendant of v in D; $high(v) = null$ if no such edge exists. These variables are useful in the applications that we mention in Appendix A. Finally, we mark each vertex v such that $(d(v), v) \in E$. For simplicity, we assume that the vertices of G are numbered from 1 to n, so we can store the above information in corresponding arrays *low*, *high*, and *mark*. Note that for a reachable vertex v, we can have $low(v) = null$ or $high(v) = null$ (or both) only if $mark(v) = true$. Before any edge insertion, all vertices are unmarked, and all entries in arrays *low* and *high* are null. We initialize the algorithm and the associated data structures by executing a linear-time algorithm to compute the dominator tree D of G [3, 10] and a linear-time algorithm to compute a low-high order δ of G [25]. So, the initialization takes $O(m + n)$ time for a digraph with n vertices and m edges.

Next, we describe the main routine, LocalInsertEdge, to handle an edge insertion. We let (x, y) be the inserted edge. Also, if x and y are reachable before the insertion, we let $z = nca(x, y)$.

From Lemmata 4 and 5 it follows that our main task now is to order the affected vertices according to a low-high order of D'. To do this, we use an auxiliary flow graph $G_A = (V_A, E_A, z)$, with start vertex z, which we refer to as the *derived affected flow graph*. Flow graph G_A is essentially a contracted version of the derived flow graph G'_z (i.e., the derived graph of z after the insertion) as we explain later. The vertices of the derived affected flow graph G_A are z, the affected vertices of G, their common ancestor c in D that is a child

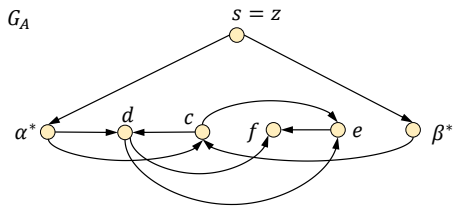

Figure 3 The derived affected flow graph G_A that corresponds to the flow graph of Figure 1 after the insertion of edge (g, d).

of z (from Lemma 3), and two auxiliary vertices α^* and β^*. Vertex α^* (resp., β^*) represents vertices in $C(z)$ with lower (resp., higher) order in δ than c. We include in G_A the edges (z, α^*) and (z, β^*). If c is marked then we include the edge (z, c) into G_A, otherwise we add the edges (α^*, c) and (β^*, c) into G_A. Also, for each edge (u, c) such that u is a descendant of an affected vertex v, we add in G_A the edge (v, c). Now we specify the edges that enter an affected vertex w in G_A. We consider each edge $(u, w) \in E$ entering w in G. We have the following cases:

(a) If u is a descendant of an affected vertex v, we add in G_A the edge (v, w).

(b) If u is a descendant of c but not a descendant of an affected vertex, then we add in G_A the edge (c, w).

(c) If $u \neq z$ is not a descendant of c, then we add the edge (α^*, w) if $u <_\delta c$, or the edge (β^*, w) if $c <_\delta u$.

(d) Finally, if $u = z$, then we add the edge (z, w). (In cases (c) and (d), $u = x$ and $w = y$.)

Recall that α (resp., β) is the sibling of c in D immediately before (resp., after) c in δ, if it exists. Then, we can obtain G_A from G'_z by contracting all vertices v with $v <_\delta c$ into $\alpha = \alpha^*$, and all vertices v with $c <_\delta v$ into $\beta = \beta^*$.

▶ **Lemma 6.** *The derived affected flow graph $G_A = (V_A, E_A, z)$ has flat dominator tree.*

Proof. We claim that for any two distinct vertices $v, w \in V_A \setminus z$, v does not dominate w. The lemma follows immediately from this claim. The claim is obvious for $w \in \{\alpha^*, \beta^*\}$, since G_A contains the edges (z, α^*) and (z, β^*). The same holds for $w = c$, since G_A contains the edge (z, c), or both the edges (α^*, c) and (β^*, c). Finally, suppose $w \in V_A \setminus \{z, \alpha^*, \beta^*\}$. Then, by the construction of G_A, vertex w is affected. By Lemma 3, $w \in D(c)$, which implies that there is a path in G from c to w that contains only vertices in $D(c)$. Hence, by construction, G_A contains a path from c to w that avoids α^* and β^*, so α^* and β^* do not dominate w. It remains to show that w is not dominated in G_A by c or another affected vertex v. Let (x, y) be the inserted edge. Without loss of generality, assume that $c <_\delta x$. Since w is affected, there is a path π in G from y to w that satisfies Lemma 1. Then π does not contain any vertex in $D[c, d(w)]$. Also, by the construction of G_A, π corresponds to a path π_A in G_A from β^* to y that avoids any vertex in $A \cap D[c, d(w)]$. Hence, w is not dominated by any vertex in $A \cap D[c, d(w)]$. It remains to show that w is not dominated by any affected vertex v in $A \setminus D[c, d(w)]$. Since both v and w are in $D(c)$ and v is not an ancestor of w in D, there is a path π' in G from c to w that contains only vertices in $D(c) \setminus \{v\}$. Then, by the construction of G_A, π' corresponds to a path π'_A in G_A from c to w that avoids v. Thus, v does not dominate w in G_A. ◀

▶ **Lemma 7.** *Let ν and μ, respectively, be the number of scanned vertices and their adjacent edges. Then, the derived affected flow graph G_A has $\nu + 4$ vertices, at most $\mu + 5$ edges, and can be constructed in $O(\nu + \mu)$ time.*

Proof. The bound on the number of vertices and edges in G_A follows from the definition of the derived affected flow graph. Next, we consider the construction time of G_A. Consider the edges entering the affected vertices. Let w be an affected vertex, and let $(u, w) \neq (x, y)$ be an edge of G'. Let q be nearest ancestor u in $C'(z)$. We distinguish two cases:

- u is not scanned. In this case, we argue that $q = c$. Indeed, it follows from the parent property of D and Lemma 3 that both u and w are descendants of c in D. Since u is not scanned, no ancestor of u in D is affected, so u remains a descendant of c in D'. Thus, $q = c$.
- u is scanned. Then, by Lemma 2, q is the nearest affected ancestor of u in D.

So we can construct the edges entering the affected vertices in G_A in two phases. In the first phase we traverse the descendants of each affected vertex q in D'. At each descendant u of q, we examine the edges leaving u. When we find an edge (u, w) with w affected, then we insert into G_A the edge (q, w). In the second phase we examine the edges entering each affected vertex w. When we find an edge (u, w) with u not visited during the first phase (i.e., u was not scanned during the update of D), we insert into G_A the edge (c, w). Note that during this construction we may insert the same edge multiple times, but this does not affect the correctness or running time of our overall algorithm. Since the descendants of an affected vertex are scanned, it follows that each phase runs in $O(\nu + \mu)$ time.

Finally, we need to consider the inserted edge (x, y). Let f be the nearest ancestor of x that is in $C(z)$. Since y is affected, $c \neq f$. Hence, we insert into G_A the edge (β^*, y) if $c <_\delta f$, and the edge (α^*, y) if $f <_\delta c$. Note that f is found during the computation of $z = nca(x, y)$, so this test takes constant time. ◀

Next, we order the vertices in $C'(z)$ according to a low-high order of ζ of G_A as follows. After computing G_A, we construct two divergent spanning trees B_A and R_A of G_A. For each vertex $v \neq z$, if (z, v) is an edge of G_A, we replace the parent of v in B_A and in R_A, denoted by $b_A(v)$ and $r_A(v)$, respectively, by z. We can compute a low-high order ζ of G_A by applying a slightly modified version of a linear-time algorithm of [25, Section 6.1] to compute a low-high order. Our modified version computes a low-high order ζ of G_A that is an augmentation of δ_z. To obtain such a low-high order, we need to assign to α^* the lowest number in ζ and to β^* the highest number in ζ. The algorithm works as follows. While G_A contains at least four vertices, we choose a vertex $v \notin \{\alpha^*, \beta^*\}$ whose in-degree in G_A exceeds its number of children in B_A plus its number of children in R_A and remove it from G_A. (From this choice of v we also have that $v \neq z$.) Then we compute recursively a low-high order for the resulting flow graph, and insert v in an appropriate location, defined by $b_A(v)$ and $r_A(v)$.

The correctness of algorithm LocalInsertEdge follows from Lemmata 4, 5 and 6. Also, by using Lemma 7, we can show that the total running time of the algorithm is bounded by $O(mn)$.

3.3 Representation of a low-high order

We consider two main options for representing a low-high order. The most straightforward is to maintain it as a preorder numbering of D, by assigning a preorder number from 1 to n to each vertex. Another option is to use a data structure for the dynamic list order problem [8, 14]. We experimented with various implementations of dynamic list order

Table 1 Real-world graphs with timestamped edges, from the Koblenz Network Collection [33].

Graph	nodes	reachable nodes	edges	avg. degree	type
temporalGraph	2029	1281	5517	2.72	Democratic National Council emails
opsahl-ucsocial	1899	1854	20296	10.69	UC Irvine messages
chess	7301	6312	60046	8.22	Chess games
munmun_digg_reply	30398	13471	85247	2.8	Digg replies
elec	7115	2316	103617	14.56	Wikipedia elections
slashdot-threads	51083	18851	130370	2.55	Slashdot threads

data structures, and in our experiments the best performance was achieved by a two-level numbering scheme that supports insertions, deletions and order queries in constant amortized time [8]. We remark that these operations suffice in all applications of our incremental algorithm that we mention in Appendix A.

Both of the above options suffices to have an implementation of the sparsification algorithm and of the local low-high order algorithm that run in total $O(mn)$ time. Since the sparsification algorithm computes the complete low-high order of a sparse subgraph of G after each update, there is no gain in using the more sophisticated numbering scheme that the dynamic list order data structure applies. For our local low-high order algorithm, however, the representation we choose is crucial for the practical performance of the algorithm. Specifically, using a dynamic list order data structure allows us to update the low-high order in amortized time proportional to the number of scanned vertices. The simple preorder numbering scheme, on the other hand, may need to renumber $O(n)$ vertices after a single update. Indeed, our preliminary experimental results confirmed that that the implementation that employs a dynamic list order data structure has superior performance.

3.4 Handling unreachable vertices

Now we provide some details on how our algorithms handle insertions of edges (x, y) when $x \in V_r$ and $y \notin V_r$, i.e., when only x is reachable from s before the insertion. In order to achieve $O(mn)$ total running time, we can simply recompute a low-high order from scratch after each such an insertion. This follows from the fact that there are at most $n - 1$ such insertions, and that we can recompute a low-high order in linear time when this type of an insertion occurs.

An alternative method, that performs much better in practice, is to compute the dominator tree and a low-high order for the vertices that were reachable from y but not from s before the insertion. Specifically, let Y be this set of vertices, and let $G[Y]$ be the flow graph with start vertex y that is induced by Y. Then, to handle the insertion of (x, y) we execute the following steps:

1. Compute the dominator tree D_Y of $G[Y]$ and a low-high order of it.
2. Link the dominator tree D of G with D_Y by making y a child of x in D, and merge appropriately the corresponding low-high orders.
3. Compute the set of edges E_Y from Y to V_r. Process each such edge as a regular insertion.

Note that after Step 2, D is the correct dominator tree for $G \setminus E_Y$ and we also have a valid low-high order of it. The insertion of the edges $(u, v) \in E_Y$ is handled as a regular insertion since both u and v are reachable from s after Step 2. In terms of running time, Steps 1 and 2 take $O(m)$ time. Also, since in Step 3 we have regular insertions, the total running time remains $O(mn)$.

Table 2 Real-world graphs used in the experiments, sorted by the file size of their largest strongly connected component (SCC). In our experiments we used both the largest SCC and the some of the 2-vertex-connected subgraphs (2VCSs), found inside the largest SCC.

Graph	Largest SCC			2VCSs			Type
	n	m	avg. degree	n	m	avg. degree	
rome99	3352	8855	2.64	2249	6467	2.88	road network [13]
twitter-higgs-retweet	13086	63537	4.86	1099	9290	8.45	twitter [35]
enron	8271	147353	17.82	4441	123527	27.82	enron mails [35]
web-NotreDame	48715	267647	5.49	1409	6856	4.87	web [35]
				1462	7279	4.98	
				1416	13226	9.34	
soc-Epinions1	32220	442768	13.74	17117	395183	23.09	trust network [35]
Amazon-302	241761	1131217	4.68	55414	241663	4.36	co-purchase [35]
WikiTalk	111878	1477665	13.21	49430	1254898	25.39	Wiki communications [35]
web-Stanford	150475	1576157	10.47	5179	129897	25.08	web [35]
				10893	162295	14.90	
web-Google	434818	3419124	7.86	77480	840829	10.85	web [35]
Amazon-601	395230	3301051	8.35	276049	2461072	8.92	co-purchase [35]
web-BerkStan	334857	4523232	13.51	1106	8206	7.42	web [35]
				4927	28142	5.71	
				12795	347465	27.16	
				29145	439148	15.07	

4 Empirical Analysis

We wrote our implementations in `C++`, using `g++ v.4.6.4` with full optimization (flag `-O3`) to compile the code. We report the running times on a GNU/Linux machine, with Ubuntu (12.04LTS): a Dell PowerEdge R715 server 64-bit NUMA machine with four AMD Opteron 6376 processors and 128GB of RAM memory. Each processor has 8 cores sharing a 16MB L3 cache, and each core has a 2MB private L2 cache and 2300MHz speed. In our experiments we did not use any parallelization, and each algorithm ran on a single core. We report CPU times measured with the `getrusage` function, averaged over ten different runs. In Tables 1 and 2 we can see some statistics about the real-world graphs we used in our experimental evaluation. In all test instances we select the first vertex of the graph as the start vertex. (Choosing a random start vertex produces similar results.) Note that the graphs in Table 1 are not strongly connected so we also report the number of vertices that are reachable from the start vertex.

The graphs in Table 1 have timestamps that indicate the moment that each edge was inserted into the graph. Thus, in our experiments, the edges are inserted according to these timestamps. The number of edges that are actually inserted is controlled by a parameter $i \in [0, 1]$ as follows. Let m be the total number of edges in the graph. Then the flow graph initially has $m - i \cdot m$ edges, and $i \cdot m$ edges are inserted one at a time. The algorithms compute (in static mode) the dominator tree and a low-high order for the first $m - i \cdot m$ edges in the original graph file and then they run in incremental mode. Note that during the execution of the algorithms some vertices may be unreachable at first and become reachable after some insertions.

We use the graphs in Table 2 to create different types of inputs by extracting their largest strongly connected component and some large 2-vertex-connected subgraphs. (A 2-vertex-connected graph remains strongly connected after the deletion of any single vertex.) We use strongly connected graphs to guarantee that all vertices are reachable from any arbitrary start vertex. Also, the 2-vertex-connected graphs are interesting because they have flat dominator trees, so inserting their edges may cause the incremental algorithms to

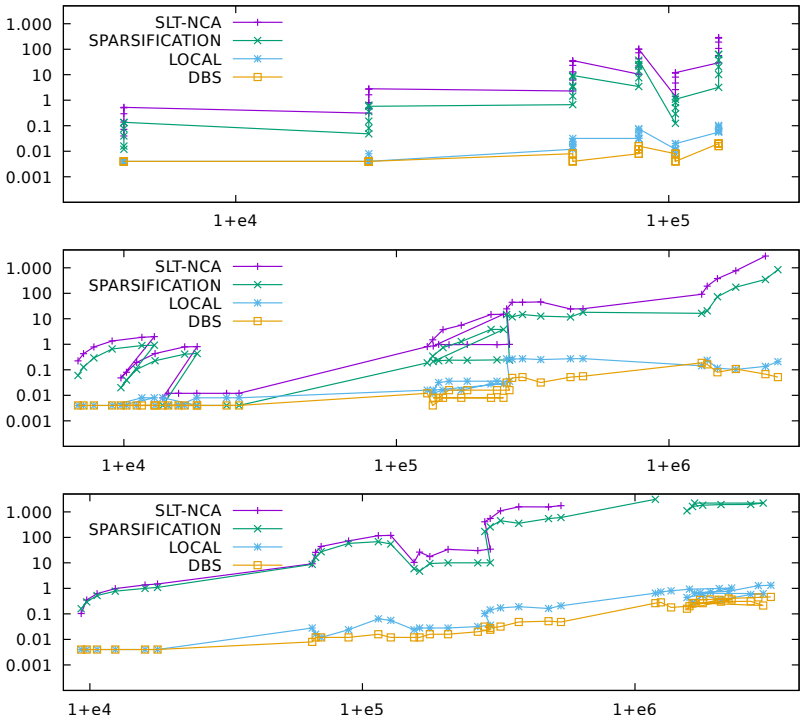

Figure 4 Incremental low-high order: timestamped graphs of Table 1 (top), random edge permutation of 2-vertex-connected graphs of Table 2 (middle), and random edge insertion in strongly connected graphs of Table 2 (bottom). Running times, in seconds, and number of edges both shown in logarithmic scale. For each input graph and each algorithm, we show the running times for inserting 5%, 10%, 20%, 40%, 80% and 100% of the edges.

perform a lot of work. We note that the graphs in Table 2 do not have timestamps, so we consider two different methods to produce dynamic graphs.

- **Random permutation:** We perform a random permutation of the edges and use the resulting order as timestamps.
- **Random insertions:** We insert $i \cdot m$ random edges in the original graph. The endpoints of each new edge are selected uniformly at random, and the edge is inserted if it is not a loop and is not already present in the current graph. Hence, the final graph has $m + i \cdot m$ edges.

We apply the first method to the 2-vertex-connected graphs, and the second method to the strongly connected graphs of Table 2. Note that in the case of random insertions the graph is strongly connected throughout the execution of the incremental algorithms. We do not apply the random insertions method to 2-vertex-connected graphs, since any edge insertion in such a graph has no effect on the dominator tree (so also the low-high order does not change).

We compare the performance of four algorithms. As a baseline, we use a static low-high order algorithm from [25] based on an efficient implementation of the Lengauer-Tarjan algorithm for computing dominators [34] from [27]. Our baseline algorithm, SLT-NCA, constructs, as intermediary, two divergent spanning trees. After each insertion of an edge (x, y), SLT-NCA tests if the insertion of (x, y) affects the dominator tree by computing the nearest common ancestor of x and y. If this is the case, then it recomputes a low-high order. The other two algorithms are the ones we presented in Section 3. For our sparsification

Table 3 Running times of the plot shown in Figure 4 (top): timestamped graphs of Table 1.

Graph	nodes	starting edges	final edges	SLT-NCA	SPARSIFICATION	LOCAL	DBS
temporalGraph05	2029	5241	5517	0.04	0.012	0.004	0.004
temporalGraph10	2029	4965	5517	0.072	0.016	0.004	0.004
temporalGraph20	2029	4414	5517	0.14	0.04	0.004	0.004
temporalGraph40	2029	3310	5517	0.296	0.076	0.004	0.004
temporalGraph80	2029	1103	5517	0.504	0.132	0.004	0.004
temporalGraph100	2029	0	5517	0.524	0.136	0.004	0.004
opsahl-ucsocial05	1899	19281	20296	0.312	0.048	0.004	0.004
opsahl-ucsocial10	1899	18266	20296	0.48	0.08	0.004	0.004
opsahl-ucsocial20	1899	16237	20296	0.812	0.152	0.004	0.004
opsahl-ucsocial40	1899	12178	20296	1.64	0.36	0.004	0.004
opsahl-ucsocial80	1899	4059	20296	2.68	0.64	0.008	0.004
opsahl-ucsocial100	1899	0	20296	2.796	0.576	0.004	0.004
chess05	7301	57044	60046	2.304	0.668	0.012	0.008
chess10	7301	54041	60046	6.14	1.456	0.016	0.008
chess20	7301	48037	60046	12.984	3.244	0.016	0.008
chess40	7301	36028	60046	23.28	3.64	0.024	0.004
chess80	7301	12009	60046	32.956	8.376	0.024	0.004
chess100	7301	0	60046	35.744	9.336	0.032	0.004
munmun_digg_reply05	30398	80985	85247	10.428	3.436	0.032	0.008
munmun_digg_reply10	30398	76722	85247	22.048	7.508	0.032	0.016
munmun_digg_reply20	30398	68198	85247	41.928	14.048	0.032	0.008
munmun_digg_reply40	30398	51148	85247	72.28	25.408	0.048	0.012
munmun_digg_reply80	30398	17049	85247	100.56	21.964	0.072	0.012
munmun_digg_reply100	30398	0	85247	100.98	37.156	0.076	0.016
elec05	7115	98436	103617	1.408	0.124	0.012	0.008
elec10	7115	93255	103617	2.62	0.292	0.012	0.004
elec20	7115	82894	103617	4.732	0.508	0.008	0.008
elec40	7115	62170	103617	8.08	0.812	0.012	0.004
elec80	7115	20723	103617	11.52	1.364	0.02	0.008
elec100	7115	0	103617	12.068	1.096	0.02	0.004
slashdot-threads05	51083	123852	130370	29.412	3.2	0.056	0.02
slashdot-threads10	51083	117333	130370	56.112	10.184	0.06	0.02
slashdot-threads20	51083	104296	130370	106.772	20.144	0.06	0.02
slashdot-threads40	51083	78222	130370	189.712	37.996	0.064	0.016
slashdot-threads80	51083	26074	130370	287.912	62.66	0.092	0.02
slashdot-threads100	51083	0	130370	270.356	62.532	0.104	0.016

algorithm of Section 3.1, denoted as SPARSIFICATION, we extend the incremental dominators algorithm DBS of [22] with the computation of two divergent spanning trees and a low-high order. Algorithm SPARSIFICATION applies these computations on a sparse subgraph of the input digraph that maintains the same dominators. Finally, we tested an implementation of our more efficient algorithm of Section 3.2, denoted as LOCAL, that updates the low-high order by computing a local low-high order of an auxiliary graph. We include also the original DBS of [22] in the experiments, to provide a more complete picture of the effectiveness of these approaches.

We compared the above incremental low-high order algorithms in three different field tests, as mentioned above. The first one, shown in Figure 4 (top) and Table 3, compares the running times of the algorithms against the dataset detailed in Table 1, i.e. the timestamped graphs. The algorithms are well distinguished: our more efficient algorithm, LOCAL, performs very well. Indeed, its running time is very close to DBS that only updates the dominator tree. Algorithm SPARSIFICATION is not competitive with LOCAL, despite the fact that it exhibits a substantial improvement over our baseline algorithm SLT-NCA.

The second experiment, shown in Figure 4 (middle) and Table 4, deals with the random permutations of the edges of 2-vertex-connected graphs. (Refer to Table 2.) As with the timestamped graphs, during the execution of the algorithms some vertices may be unreachable at first, but here all vertices become reachable in the end. Also, at the end of all insertions, the final graph has flat dominator tree. Here we can see that, as before, the algorithms are still distinguished, but in this case the two couples SPARSIFICATION and SLT-NCA, and LOCAL and DBS, are closer.

Table 4 Running times of the plot shown in Figure 4 (middle): random edge permutation of 2-vertex-connected graphs of Table 2.

Graph	nodes	starting edges	final edges	SLT-NCA	SPARSIFICATION	LOCAL	DBS
rome05	2249	6467	6790	0.224	0.06	0.004	0.004
rome10	2249	6467	7114	0.428	0.128	0.004	0.004
rome20	2249	6467	7760	0.784	0.292	0.004	0.004
rome40	2249	6467	9054	1.348	0.656	0.004	0.004
rome80	2249	6467	11641	1.868	0.896	0.008	0.004
rome100	2249	6467	12934	1.992	0.92	0.008	0.004
twitter05	1099	9290	9755	0.048	0.02	0.004	0.004
twitter10	1099	9290	10219	0.08	0.04	0.004	0.004
twitter20	1099	9290	11148	0.196	0.104	0.004	0.004
twitter40	1099	9290	13006	0.432	0.228	0.004	0.004
twitter80	1099	9290	16722	0.796	0.412	0.004	0.004
twitter100	1099	9290	18580	0.812	0.436	0.004	0.004
NotreDame05	1416	13226	13887	0.008	0.004	0.008	0.004
NotreDame10	1416	13226	14549	0.012	0.004	0.004	0.004
NotreDame20	1416	13226	15871	0.012	0.004	0.004	0.004
NotreDame40	1416	13226	18516	0.012	0.004	0.008	0.004
NotreDame80	1416	13226	23807	0.012	0.004	0.008	0.004
NotreDame100	1416	13226	26452	0.012	0.004	0.008	0.004
enron05	4441	123527	129703	0.808	0.188	0.016	0.012
enron10	4441	123527	135880	1.504	0.344	0.012	0.004
enron20	4441	123527	148232	3.744	0.748	0.016	0.008
enron40	4441	123527	172938	5.584	1.28	0.016	0.008
enron80	4441	123527	222349	14.744	3.836	0.028	0.008
enron100	4441	123527	247054	15.076	3.828	0.028	0.008
webStanford05	5179	129897	136392	0.856	0.212	0.016	0.008
webStanford10	5179	129897	142887	0.992	0.228	0.032	0.008
webStanford20	5179	129897	155876	0.964	0.24	0.036	0.016
webStanford40	5179	129897	181856	0.98	0.236	0.036	0.016
webStanford80	5179	129897	233815	0.968	0.244	0.036	0.016
webStanford100	5179	129897	259794	0.988	0.24	0.036	0.016
Amazon05	55414	241663	253746	24.528	14.592	0.268	0.032
Amazon10	55414	241663	265829	44.392	11.88	0.264	0.048
Amazon20	55414	241663	289996	44.356	14.704	0.272	0.052
Amazon40	55414	241663	338328	45.792	12.628	0.252	0.032
Amazon80	55414	241663	434993	24.22	11.82	0.272	0.052
Amazon100	55414	241663	483326	24.856	18.096	0.276	0.056
WikiTalk05	49430	1254898	1317643	91.808	16.344	0.144	0.188
WikiTalk10	49430	1254898	1380388	190.616	20.42	0.236	0.164
WikiTalk20	49430	1254898	1505878	374.292	73.252	0.116	0.08
WikiTalk40	49430	1254898	1756857	766.28	172.064	0.104	0.108
WikiTalk80	49430	1254898	2258816	2868.28	349.632	0.136	0.068
WikiTalk100	49430	1254898	2509796	> 3600	837.728	0.208	0.052

The last experiment, detailed in Figure 4 (bottom) and Table 5, concerns the random edge insertion in strongly connected graphs of Table 2. The ranking of the algorithms does not change, as we can see in Figure 4 (bottom), but the difference is bigger: we note a bigger gap of more than two orders of magnitude, in particular, between LOCAL and the couple SLT-NCA and SPARSIFICATION.

From all the above experimental results, it is evident that a careful implementation of our efficient algorithm LOCAL has excellent performance in practice. Indeed, its running time is very close to the running time of an efficient incremental algorithm for updating the dominator tree.

Table 5 Running times of the plot shown in Figure 4 (bottom): random edge insertion in strongly connected graphs of Table 2.

Graph	nodes	starting edges	final edges	SLT-NCA	SPARSIFICATION	LOCAL	DBS
rome05	3352	8855	9298	0.104	0.16	0.004	0.004
rome10	3352	8855	9741	0.352	0.296	0.004	0.004
rome20	3352	8855	10626	0.624	0.528	0.004	0.004
rome40	3352	8855	12397	0.98	0.78	0.004	0.004
rome80	3352	8855	15939	1.372	1	0.004	0.004
rome100	3352	8855	17710	1.48	1.088	0.004	0.004
twitter05	13086	63537	65444	9.252	8.74	0.028	0.008
twitter10	13086	63537	67344	25.716	16.452	0.016	0.012
twitter20	13086	63537	70544	44.148	27.124	0.012	0.012
twitter40	13086	63537	88952	72.732	57.828	0.024	0.012
twitter80	13086	63537	114367	116.452	68.1	0.064	0.016
twitter100	13086	63537	127074	119.152	54.624	0.056	0.012
enron05	8271	147353	154721	10.52	5.928	0.024	0.012
enron10	8271	147353	162088	26.512	4.712	0.028	0.012
enron20	8271	147353	176824	17.652	9.4	0.028	0.016
enron40	8271	147353	206294	33.724	10.044	0.028	0.016
enron80	8271	147353	265235	30.34	10.02	0.032	0.02
enron100	8271	147353	294706	34.496	10.012	0.036	0.024
NotreDame05	48715	267647	281029	409.072	169.168	0.104	0.032
NotreDame10	48715	267647	294412	550.444	259.42	0.144	0.028
NotreDame20	48715	267647	321176	1093.96	447.44	0.172	0.032
NotreDame40	48715	267647	374706	1575.83	356.588	0.192	0.048
NotreDame80	48715	267647	481765	1563.68	544.748	0.164	0.052
NotreDame100	48715	267647	535294	1753.4	597.776	0.208	0.048
Amazon05	241761	1131217	1187778	> 3600	3098.83	0.652	0.264
Amazon10	241761	1131217	1244339	> 3600	> 3600	0.732	0.284
Amazon20	241761	1131217	1357460	> 3600	> 3600	0.804	0.18
Amazon40	241761	1131217	1583704	> 3600	> 3600	0.936	0.2
Amazon80	241761	1131217	2036191	> 3600	> 3600	0.992	0.36
Amazon100	241761	1131217	2262434	> 3600	> 3600	1.032	0.368
WikiTalk05	111878	1477665	1551548	> 3600	1096.12	0.44	0.16
WikiTalk10	111878	1477665	1625432	> 3600	1619.04	0.28	0.264
WikiTalk20	111878	1477665	1773198	> 3600	1831.12	0.544	0.264
WikiTalk40	111878	1477665	2068731	> 3600	1932.21	0.36	0.304
WikiTalk80	111878	1477665	2659797	> 3600	1947	0.576	0.312
WikiTalk100	111878	1477665	2955330	> 3600	2207.42	0.62	0.212
webStanford05	150475	1576157	1654965	> 3600	2208.26	0.648	0.268
webStanford10	150475	1576157	1733773	> 3600	> 3600	0.676	0.356
webStanford20	150475	1576157	1891388	> 3600	> 3600	0.732	0.372
webStanford40	150475	1576157	2206620	> 3600	> 3600	0.768	0.404
webStanford80	150475	1576157	2837083	> 3600	> 3600	1.288	0.444
webStanford100	150475	1576157	3152314	> 3600	> 3600	1.324	0.464

References

1. A. Abboud and V. Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *Proc. 55th IEEE Symposium on Foundations of Computer Science, FOCS*, pages 434–443, 2014. doi:10.1109/FOCS.2014.53.
2. S. Allesina and A. Bodini. Who dominates whom in the ecosystem? Energy flow bottlenecks and cascading extinctions. *Journal of Theoretical Biology*, 230(3):351–358, 2004.
3. S. Alstrup, D. Harel, P. W. Lauridsen, and M. Thorup. Dominators in linear time. *SIAM Journal on Computing*, 28(6):2117–32, 1999.
4. S. Alstrup and P. W. Lauridsen. A simple dynamic algorithm for maintaining a dominator tree. Technical Report 96-3, Department of Computer Science, University of Copenhagen, 1996.
5. M. E. Amyeen, W. K. Fuchs, I. Pomeranz, and V. Boppana. Fault equivalence identification using redundancy information and static and dynamic extraction. In *Proceedings of the 19th IEEE VLSI Test Symposium*, March 2001.
6. S. Baswana, K. Choudhary, and L. Roditty. Fault tolerant reachability for directed graphs. In Yoram Moses, editor, *Distributed Computing*, volume 9363 of *Lecture Notes in Computer Science*, pages 528–543. Springer Berlin Heidelberg, 2015. doi:10.1007/978-3-662-48653-5_35.
7. S. Baswana, K. Choudhary, and L. Roditty. Fault tolerant reachability subgraph: Generic and optimal. In *Proc. 48th ACM Symp. on Theory of Computing*, pages 509–518, 2016. doi:10.1145/2897518.2897648.
8. M. A. Bender, R. Cole, E. D. Demaine, M. Farach-Colton, and J. Zito. Two simplified algorithms for maintaining order in a list. In *Proceedings of the 10th Annual European Symposium on Algorithms*, pages 152–164, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. doi:10.1007/3-540-45749-6_17.
9. M. A. Bender, J. T. Fineman, S. Gilbert, and R. E. Tarjan. A new approach to incremental cycle detection and related problems. *ACM Transactions on Algorithms*, 12(2):14:1–14:22, December 2015. doi:10.1145/2756553.
10. A. L. Buchsbaum, L. Georgiadis, H. Kaplan, A. Rogers, R. E. Tarjan, and J. R. Westbrook. Linear-time algorithms for dominators and other path-evaluation problems. *SIAM Journal on Computing*, 38(4):1533–1573, 2008.
11. S. Cicerone, D. Frigioni, U. Nanni, and F. Pugliese. A uniform approach to semi-dynamic problems on digraphs. *Theor. Comput. Sci.*, 203:69–90, August 1998.
12. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991. doi:10.1145/115372.115320.
13. C. Demetrescu, A. V. Goldberg, and D. S. Johnson. 9th DIMACS Implementation Challenge: Shortest Paths. 2007. URL: http://www.diag.uniroma1.it/challenge9/.
14. P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *Proc. 19th ACM Symp. on Theory of Computing*, pages 365–372, 1987.
15. W. Fraczak, L. Georgiadis, A. Miller, and R. E. Tarjan. Finding dominators via disjoint set union. *Journal of Discrete Algorithms*, 23:2–20, 2013. doi:http://dx.doi.org/10.1016/j.jda.2013.10.003.
16. H. N. Gabow. The minset-poset approach to representations of graph connectivity. *ACM Transactions on Algorithms*, 12(2):24:1–24:73, February 2016. doi:10.1145/2764909.
17. K. Gargi. A sparse algorithm for predicated global value numbering. *SIGPLAN Not.*, 37(5):45–56, May 2002. doi:10.1145/543552.512536.
18. L. Georgiadis. Testing 2-vertex connectivity and computing pairs of vertex-disjoint *s*-*t* paths in digraphs. In *Proc. 37th Int'l. Coll. on Automata, Languages, and Programming*, pages 738–749, 2010.

19 L. Georgiadis. Approximating the smallest 2-vertex connected spanning subgraph of a directed graph. In *Proc. 19th European Symposium on Algorithms*, pages 13–24, 2011.

20 L. Georgiadis, G. F. Italiano, L. Laura, and N. Parotsidis. 2-vertex connectivity in directed graphs. In *Proc. 42nd Int'l. Coll. on Automata, Languages, and Programming*, pages 605–616, 2015. doi:10.1007/978-3-662-47672-7_49.

21 L. Georgiadis, G. F. Italiano, L. Laura, and N. Parotsidis. 2edgee connectivity in directed graphs. *ACM Trans. Algorithms*, 13(1):9:1–9:24, October 2016. doi:10.1145/2968448.

22 L. Georgiadis, G. F. Italiano, L. Laura, and F. Santaroni. An experimental study of dynamic dominators. In *Proc. 20th European Symposium on Algorithms*, pages 491–502, 2012. Full version: *CoRR*, abs/1604.02711.

23 L. Georgiadis, G. F. Italiano, and N. Parotsidis. Incremental 2-edge-connectivity in directed graphs. In *Proc. 43rd Int'l. Coll. on Automata, Languages, and Programming*, pages 49:1–49:15, 2016. doi:10.4230/LIPIcs.ICALP.2016.49.

24 L. Georgiadis, A. Karanasiou, G. Konstantinos, and L. Laura. On low-high orders of directed graphs: Incremental algorithms and applications. *CoRR*, abs/1608.06462, 2016. URL: http://arxiv.org/abs/1608.06462.

25 L. Georgiadis and R. E. Tarjan. Dominator tree certification and divergent spanning trees. *ACM Transactions on Algorithms*, 12(1):11:1–11:42, November 2015. doi:10.1145/2764913.

26 L. Georgiadis and R. E. Tarjan. Addendum to "Dominator tree certification and divergent spanning trees". *ACM Transactions on Algorithms*, 12(4):56:1–56:3, August 2016. doi:10.1145/2928271.

27 L. Georgiadis, R. E. Tarjan, and R. F. Werneck. Finding dominators in practice. *Journal of Graph Algorithms and Applications (JGAA)*, 10(1):69–94, 2006.

28 M. Gomez-Rodriguez and B. Schölkopf. Influence maximization in continuous time diffusion networks. In *29th International Conference on Machine Learning (ICML)*, 2012.

29 M. Henzinger, S. Krinninger, and V. Loitzenbauer. Finding 2-edge and 2-vertex strongly connected components in quadratic time. In *Proc. 42nd Int'l. Coll. on Automata, Languages, and Programming*, pages 713–724, 2015. doi:10.1007/978-3-662-47672-7_58.

30 G. F. Italiano, L. Laura, and F. Santaroni. Finding strong bridges and strong articulation points in linear time. *Theoretical Computer Science*, 447:74–84, 2012. doi:10.1016/j.tcs.2011.11.011.

31 R. Jaberi. Computing the 2-blocks of directed graphs. *RAIRO-Theor. Inf. Appl.*, 49(2):93–119, 2015. doi:10.1051/ita/2015001.

32 R. Jaberi. On computing the 2-vertex-connected components of directed graphs. *Discrete Applied Mathematics*, 204:164–172, 2016. doi:10.1016/j.dam.2015.10.001.

33 J. Kunegis. KONECT: the Koblenz network collection. In *22nd International World Wide Web Conference, WWW'13, Rio de Janeiro, Brazil, May 13-17, 2013, Companion Volume*, pages 1343–1350, 2013.

34 T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–41, 1979.

35 J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection, June 2014. URL: http://snap.stanford.edu/data.

36 E. K. Maxwell, G. Back, and N. Ramakrishnan. Diagnosing memory leaks using graph mining on heap dumps. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD'10, pages 115–124, 2010.

37 R. M. McConnell, K. Mehlhorn, S. Näher, and P. Schweitzer. Certifying algorithms. *Computer Science Review*, 5(2):119–161, 2011.

38 M. Mowbray and A. Lain. Dominator-tree analysis for distributed authorization. In *Proceedings of the Third ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS'08, pages 101–112, New York, NY, USA, 2008. ACM. doi:10.1145/1375696.1375709.

39 H. Nagamochi and T. Ibaraki. A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph. *Algorithmica*, 7:583–596, 1992.

40 L. Quesada, P. Van Roy, Y. Deville, and R. Collet. Using dominators for solving constrained path problems. In *Proc. 8th International Conference on Practical Aspects of Declarative Languages*, pages 73–87, 2006.

41 G. Ramalingam and T. Reps. An incremental algorithm for maintaining the dominator tree of a reducible flowgraph. In *Proc. 21st ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 287–296, 1994.

42 V. C. Sreedhar, G. R. Gao, and Y. Lee. Incremental computation of dominator trees. *ACM Transactions on Programming Languages and Systems*, 19:239–252, 1997.

43 R. E. Tarjan. Finding dominators in directed graphs. *SIAM Journal on Computing*, 3(1):62–89, 1974.

44 R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

45 R. E. Tarjan. Fast algorithms for solving path problems. *Journal of the ACM*, 28(3):594–614, 1981.

46 T. Tholey. Linear time algorithms for two disjoint paths problems on directed acyclic graphs. *Theoretical Computer Science*, 465:35–48, 2012. doi:10.1016/j.tcs.2012.09.025.

47 J. Zhao and S. Zdancewic. Mechanized verification of computing dominators for formalizing compilers. In *Proc. 2nd International Conference on Certified Programs and Proofs*, pages 27–42. Springer, 2012. doi:10.1007/978-3-642-35308-6_6.

A Applications of incremental low-high orders

In this section we show how our result on incremental low-high order maintenance implies the following incremental algorithms that also run in $O(mn)$ total time for a sequence of m edge insertions.

- First, we give an algorithm that maintains, after each edge insertion, two strongly divergent spanning trees of G, and answers the following queries in constant time: (i) For any two query vertices v and w, find a path π_{sv} from s to v and a path π_{sw} from s to w, such that π_{sv} and π_{sw} share only the common dominators of v and w. We can output these paths in $O(|\pi_{sv}| + |\pi_{sw}|)$ time. (ii) For any two query vertices v and w, find a path π_{sv} from s to v that avoids w, if such a path exists. We can output this path in $O(|\pi_{sv}|)$ time.
- Then we provide an algorithm for an incremental version of the fault-tolerant reachability problem [6, 7]. We maintain a flow graph $G = (V, E, s)$ with n vertices through a sequence of m edge insertions, so that we can answer the following query in $O(n)$ time. Given a spanning forest $F = (V, E_F)$ of G rooted at s, find a set of edges $E' \subseteq E \setminus E_F$ of minimum cardinality, such that the subgraph $G' = (V, E_F \cup E', s)$ of G has the same dominators as G.
- Finally, given a digraph G, we consider how to maintain incrementally a spanning subgraph of G with $O(n)$ edges that preserves the 2-edge-connectivity relations in G.

A.1 Strongly divergent spanning trees and path queries

We can use the arrays *mark*, *low*, and *high* to maintain a pair of strongly divergent spanning trees, B and R, of G after each update. Recall that B and R are *strongly divergent* if for every pair of vertices v and w, we have $B[s,v] \cap R[s,w] = D[s,v] \cap D[s,w]$ or $R[s,v] \cap B[s,w] = D[s,v] \cap D[s,w]$. Moreover, we can construct B and R so that they are also edge-disjoint except for the bridges of G. A *bridge* of G is an edge (u,v) that is contained in every path from s to v. Let $b(v)$ (resp., $r(v)$) denote the parent of a vertex v in B (resp., R). To update B and R after the insertion of an edge (x,y), we only need to update $b(v)$ and $r(v)$ for the affected vertices v, and possibly for their common ancestor c that is a child of $z = nca(x,y)$ from Lemma 3. We can update $b(v)$ and $r(v)$ of each vertex $v \in A \cup \{c\}$ as follows: set $b(v) \leftarrow d(v)$ if $low(v) = null$, $b(v) \leftarrow low(v)$ otherwise; set $r(v) \leftarrow d(v)$ if $high(v) = null$, $r(v) \leftarrow high(v)$ otherwise. If the insertion of (x,y) does not affect y, then $A = \emptyset$ but we may still need to update $b(y)$ and $r(y)$ if $x \notin D(y)$ in order to make B and R maximally edge-disjoint. Note that in this case $z = d(y)$, so we only need to check if both $low(y)$ and $high(y)$ are null. If they are, then we set $low(y) \leftarrow x$ if $x <_\delta y$, and set $high(y) \leftarrow x$ otherwise. Then, we can update $b(y)$ and $r(y)$ as above.

Now consider a query that, given two vertices v and w, asks for two maximally vertex-disjoint paths, π_{sv} and π_{sw}, from s to v and from s to w, respectively. Such queries were used in [46] to give a linear-time algorithm for the 2-disjoint paths problem on a directed acyclic graph. If $v <_\delta w$, then we select $\pi_{sv} \leftarrow B[s,v]$ and $\pi_{sw} \leftarrow R[s,w]$; otherwise, we select $\pi_{sv} \leftarrow R[s,v]$ and $\pi_{sw} \leftarrow B[s,w]$. Therefore, we can find such paths in constant time, and output them in $O(|\pi_{sv}| + |\pi_{sw}|)$ time. Similarly, for any two query vertices v and w, we can report a path π_{sv} from s to v that avoids w. Such a path exists if and only if w does not dominate v, which we can test in constant time using the ancestor-descendant relation in D [43]. If w does not dominate v, then we select $\pi_{sv} \leftarrow B[s,v]$ if $v <_\delta w$, and select $\pi_{sv} \leftarrow R[s,v]$ if $w <_\delta v$.

A.2 Fault tolerant reachability

Baswana et al. [6] study the following reachability problem. We are given a flow graph $G = (V,E,s)$ and a spanning tree $T = (V,E_T)$ rooted at s. We call a set of edges E' *valid* if the subgraph $G' = (V, E_T \cup E', s)$ of G has the same dominators as G. The goal is to find a valid set of minimum cardinality. As shown in [26], we can compute a minimum-size valid set in $O(m)$ time, given the dominator tree D and a low-high order of δ of it. We can combine the above construction with our incremental low-high algorithm to solve the incremental version of the fault tolerant reachability problem, where G is modified by edge insertions and we wish to compute efficiently a valid set for any query spanning tree T. Let $t(v)$ be the parent of v in T. Our algorithm maintains, after each edge insertion, a low-high order δ of G, together with the *mark*, *low*, and *high* arrays. Given a query spanning tree $T = (V,E_T)$, we can compute a valid set of minimum cardinality E' as follows. For each vertex $v \neq s$, we apply the appropriate one of the following cases: (a) If $t(v) = d(v)$ then we do not insert into E' any edge entering v. (b) If $t(v) \neq d(v)$ and v is marked then we insert $(d(v),v)$ into E'. (c) If v is not marked then we consider the following subcases: If $t(v) >_\delta v$, then we insert into E' the edge (x,v) with $x = low(v)$. Otherwise, if $t(v) <_\delta v$, then we insert into E' the edge (x,v) with $x = high(v)$. Hence, can update the minimum valid set in $O(mn)$ total time.

We note that the above construction can be easily generalized for the case where T is forest, i.e., when E_T is a subset of the edges of some spanning tree of G. In this case, $t(v)$ can be null for some vertices $v \neq s$. To answer a query for such a T, we apply the previous

construction with the following modification when $t(v)$ is null. If v is marked then we insert $(d(v), v)$ into E', as in case (b). Otherwise, we insert both edges entering v from $low(v)$ and $high(v)$. In particular, when $E_T = \emptyset$, we compute a subgraph $G' = (V, E', s)$ of G with minimum number of edges that has the same dominators as G. This corresponds to the case $k = 1$ in [7].

A.3 Sparse certificate for 2-edge-connectivity

Let $G = (V, E)$ be a strongly connected digraph. We say that vertices $u, v \in V$ are 2-*edge-connected* if there are two edge-disjoint directed paths from u to v and two edge-disjoint directed paths from v to u. (A path from u to v and a path from v to u need not be edge-disjoint.) A 2-*edge-connected block* of a digraph $G = (V, E)$ is defined as a maximal subset $B \subseteq V$ such that every two vertices in B are 2-edge-connected. If G is not strongly connected, then its 2-edge-connected blocks are the 2-edge-connected blocks of each strongly connected component of G. A *sparse certificate* for the 2-edge-connected blocks of G is a spanning subgraph $C(G)$ of G that has $O(n)$ edges and maintains the same 2-edge-connected blocks as G. Sparse certificates of this kind allow us to speed up computations, such as finding the actual edge-disjoint paths that connect a pair of vertices (see, e.g., [39]). The 2-edge-connected blocks and a corresponding sparse certificate can be computed in $O(m + n)$ time [21]. An incremental algorithm for maintaining the 2-edge-connected blocks is presented in [23]. This algorithm maintains the dominator tree of G, with respect to an arbitrary start vertex s, and of its reversal G^R, together with the auxiliary components of G and G^R, defined next.

Recall that an edge (u, v) is a *bridge* of a flow graph G with start vertex s if all paths from s to v include (u, v). After deleting from the dominator tree D the bridges of G, we obtain the *bridge decomposition* of D into a forest $\mathcal{D}$. For each root r of a tree in the bridge decomposition $\mathcal{D}$ we define the *auxiliary graph* $G_r = (V_r, E_r)$ *of* r as follows. The vertex set V_r of G_r consists of all the vertices in D_r. The edge set E_r contains all the edges of G among the vertices of V_r, referred to as *ordinary* edges, and a set of *auxiliary* edges, which are obtained by contracting vertices in $V \setminus V_r$, as follows. Let v be a vertex in V_r that has a child w in $V \setminus V_r$. Note that w is a root in the bridge decomposition $\mathcal{D}$ of D. For each such child w of v, we contract w and all its descendants in D into v. The *auxiliary components* of G are the strongly connected components of each auxiliary graph G_r.

We sketch how to extend the incremental algorithm of [23] so that it also maintains a sparse certificate $C(G)$ for the 2-edge-connected components of G, in $O(mn)$ total time. It suffices to maintain the auxiliary components in G and G^R, and two maximally edge-disjoint divergent spanning trees for each of G and G^R. We can maintain these divergent spanning trees as described in Section A.1. To identify the auxiliary components, the algorithm of [23] uses, for each auxiliary graph, an incremental algorithm for maintaining strongly connected components [9]. It is easy to extend this algorithm so that it also computes $O(n)$ edges that define these strongly connected components. The union of these edges and of the edges in the divergent spanning trees are the edges of $C(G)$.